Doing Research

- analysing problems/languages
- computability/solvability/decidability — is there an algorithm?
- computational complexity — is it practical?
- expressive power — are there things that cannot be expressed?
- formal languages provide well-studied models
Formal Languages

- given a finite *alphabet* (set) of symbols $\Sigma$
  — e.g., $\Sigma = \{0, 1\}$
- a *string* is a sequence (concatenation) of symbols
  — e.g., $0101$
- all finite strings over $\Sigma$ are denoted by $\Sigma^*$
  — e.g., $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

*Language* $L$ over $\Sigma$ is just a subset of $\Sigma^*$
— e.g., $L_1$: strings with an even number of 1’s
— e.g., $L_0$: strings representing valid Java programs
  (over an alphabet of all legal symbols in Java)

- are there finite representations for infinite languages?
Formal Languages

- given a finite *alphabet* (set) of symbols $\Sigma$
  - e.g., $\Sigma = \{0, 1\}$

- a *string* is a sequence (concatenation) of symbols
  - e.g., $0101$

- all finite strings over $\Sigma$ are denoted by $\Sigma^*$
  - e.g., $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

- *language* $L$ over $\Sigma$ is just a subset of $\Sigma^*$
  - e.g., $L_1$: strings with an even number of 1’s
  - e.g., $L_0$: strings representing valid Java programs (over an alphabet of all legal symbols in Java)

- are there finite representations for infinite languages?

- yes, *grammars* (generative) and *automata* (recognition)
Automata

- device (machine) for recognising (accepting) a language
- provide models of computation
- automaton comprises states and transitions between states
- automaton is given a string as input
- automaton $M$ accepts a string $w$ by halting in an accept state, when given $w$ as input
- language $L(M)$ accepted by automaton $M$ is the set of all strings which $M$ accepts
Types of Automata

- finite state automaton
  - deterministic
  - nondeterministic
- pushdown automaton
- linear-bounded automaton
- Turing machine
Example of a Finite State Automaton

- $L_1$ (strings with an even number of 1’s) can be recognised by the following FSA
  - 2 states $s_{even}$ and $s_{odd}$
  - 4 transitions
  - $s_{even}$ is both the initial and final state

- FSA recognises 011:
Example of a Finite State Automaton

- $L_1$ (strings with an even number of 1’s) can be recognised by the following FSA
  - 2 states $s_{even}$ and $s_{odd}$
  - 4 transitions
  - $s_{even}$ is both the initial and final state

FSA recognises 011:
Example of a Finite State Automaton

- $L_1$ (strings with an even number of 1’s) can be recognised by the following FSA
  - 2 states $s_{even}$ and $s_{odd}$
  - 4 transitions
  - $s_{even}$ is both the initial and final state

- FSA recognises 011: 0
Example of a Finite State Automaton

- $L_1$ (strings with an even number of 1’s) can be recognised by the following FSA
  - 2 states $s_{even}$ and $s_{odd}$
  - 4 transitions
  - $s_{even}$ is both the initial and final state

FSA recognises 011: 01
Example of a Finite State Automaton

- $L_1$ (strings with an even number of 1’s) can be recognised by the following FSA
  - 2 states $s_{\text{even}}$ and $s_{\text{odd}}$
  - 4 transitions
  - $s_{\text{even}}$ is both the initial and final state

- FSA recognises 011: 011
Grammars

- **grammars** generate languages using:
  - symbols from alphabet \( \Sigma \) (called *terminals*)
  - set \( N \) of *nonterminals* (one designated as *starting*)
  - set \( P \) of *productions*, each of the form
    \[ U \rightarrow V \]
    where \( U \) and \( V \) are (loosely) strings over \( \Sigma \cup N \)
  - a string (sequence of terminals) \( w \) is generated by \( G \) if there is a *derivation* of \( w \) using \( G \), starting from the *starting* nonterminal of \( G \)
  - language *generated* by grammar \( G \), denoted \( L(G) \), is the set of strings which can be derived using \( G \)
Grammar Example

- $L_1$ (strings with an even number of 1’s) can be generated by a grammar with productions

$$
\begin{align*}
S & \rightarrow \epsilon \\
S & \rightarrow 0S \\
S & \rightarrow 1T \\
T & \rightarrow 0T \\
T & \rightarrow 1S
\end{align*}
$$

where $S$ is the *starting* nonterminal
Grammar Example

- \( L_1 \) (strings with an even number of 1’s) can be generated by a grammar with productions

\[
S \rightarrow \epsilon \\
S \rightarrow 0S \\
S \rightarrow 1T \\
T \rightarrow 0T \\
T \rightarrow 1S
\]

where \( S \) is the starting nonterminal

- a derivation of 01010 is given by

\[
S \Rightarrow 0S
\]
Grammar Example

- \( L_1 \) (strings with an even number of 1’s) can be generated by a grammar with productions

\[
\begin{align*}
S & \rightarrow \epsilon \\
S & \rightarrow 0S \\
S & \rightarrow 1T \\
T & \rightarrow 0T \\
T & \rightarrow 1S
\end{align*}
\]

where \( S \) is the starting nonterminal

- a derivation of 01010 is given by

\[
S \Rightarrow 0S \Rightarrow 01T
\]
Grammar Example

- $L_1$ (strings with an even number of 1’s) can be generated by a grammar with productions

  \[
  S \rightarrow \epsilon \\
  S \rightarrow 0S \\
  S \rightarrow 1T \\
  T \rightarrow 0T \\
  T \rightarrow 1S
  \]

  where $S$ is the starting nonterminal

- a derivation of 01010 is given by

  \[
  S \Rightarrow 0S \Rightarrow 01T \Rightarrow 010T
  \]
Grammar Example

- $L_1$ (strings with an even number of 1’s) can be generated by a grammar with productions

\[
\begin{align*}
S & \rightarrow \epsilon \\
S & \rightarrow 0S \\
S & \rightarrow 1T \\
T & \rightarrow 0T \\
T & \rightarrow 1S 
\end{align*}
\]

where $S$ is the *starting* nonterminal

- a *derivation* of $01010$ is given by

\[
S \Rightarrow 0S \Rightarrow 01T \Rightarrow 010T \Rightarrow 0101S
\]
Grammar Example

- $L_1$ (strings with an even number of 1’s) can be generated by a grammar with productions

  \[
  S \rightarrow \epsilon \\
  S \rightarrow 0S \\
  S \rightarrow 1T \\
  T \rightarrow 0T \\
  T \rightarrow 1S
  \]

  where $S$ is the starting nonterminal

- a derivation of $01010$ is given by

  \[
  S \Rightarrow 0S \Rightarrow 01T \Rightarrow 010T \Rightarrow 0101S \Rightarrow 01010S
  \]
Grammar Example

- \( L_1 \) (strings with an even number of 1’s) can be generated by a grammar with productions:

\[
\begin{align*}
S & \rightarrow \epsilon \\
S & \rightarrow 0S \\
S & \rightarrow 1T \\
T & \rightarrow 0T \\
T & \rightarrow 1S
\end{align*}
\]

where \( S \) is the starting nonterminal.

- a derivation of 01010 is given by:

\[
S \Rightarrow 0S \Rightarrow 01T \Rightarrow 010T \Rightarrow 0101S \Rightarrow 01010S \Rightarrow 01010
\]
Uses of Grammars

- to specify syntax of programming languages
- in natural language understanding
- in pattern recognition
- to specify schemas (types) for tree-structured data, e.g., XML
- ...
Restrictions on productions give different types of grammars:
- **Regular** (type 3)
- **Context-free** (type 2)
- **Context-sensitive** (type 1)
- **Phrase-structure** (type 0)

- For context-free, e.g., left side must be single nonterminal
- No restrictions for phrase-structure
- Language is of type $i$ iff there is a grammar of type $i$ which generates it
Examples of Language Hierarchy

- varying expressive power
- regular $\subset$ context-free $\subset$ context-sensitive $\subset$ phrase-structure
Examples of Language Hierarchy

- varying expressive power
- regular $\subset$ context-free $\subset$ context-sensitive $\subset$ phrase-structure
- $L_1$ (strings over $\{0, 1\}$ with an even number of 1’s) is regular
Examples of Language Hierarchy

- varying expressive power
- regular $\subset$ context-free $\subset$ context-sensitive $\subset$ phrase-structure
- $L_1$ (strings over $\{0, 1\}$ with an even number of 1’s) is regular
- $L_2 = \{0^n1^n \mid n \geq 0\}$ is context-free, but not regular
Examples of Language Hierarchy

- varying expressive power
- regular $\subset$ context-free $\subset$ context-sensitive $\subset$ phrase-structure
- $L_1$ (strings over $\{0, 1\}$ with an even number of 1’s) is regular
- $L_2 = \{0^n1^n \mid n \geq 0\}$ is context-free, but not regular
- $L_3 = \{ww \mid w \in \{0, 1\}^*\}$ is context-sensitive, but not context-free
Examples of Language Hierarchy

- varying expressive power
- regular $\subset$ context-free $\subset$ context-sensitive $\subset$ phrase-structure
- $L_1$ (strings over $\{0, 1\}$ with an even number of 1’s) is regular
- $L_2 = \{0^n1^n | n \geq 0\}$ is context-free, but not regular
- $L_3 = \{ww | w \in \{0, 1\}^*\}$ is context-sensitive, but not context-free
- there exists a phrase-structure (recursive) language which is not context-sensitive
Complexity of Grammar Problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is $w \in L(G)$?</td>
<td>P</td>
</tr>
<tr>
<td>Is $L(G)$ empty?</td>
<td>P</td>
</tr>
<tr>
<td>Is $L(G_1) \equiv L(G_2)$?</td>
<td>PSPACE</td>
</tr>
</tbody>
</table>

- P: decidable in polynomial time
- PSPACE: decidable in polynomial space (and complete for PSPACE: at least as hard as NP-complete)
- U: undecidable
- so type of grammar has significant effect on complexity
A language is

\[
\begin{align*}
\text{regular} & \iff \text{accepted by a finite-state automaton} \\
\text{context-free} & \iff \text{accepted by a pushdown automaton} \\
\text{context-sensitive} & \iff \text{accepted by a linear-bounded automaton} \\
\text{phrase-structure} & \iff \text{accepted by a Turing machine}
\end{align*}
\]
Regular Expressions

- algebraic notation for denoting regular languages
- use $\circ$ (concatenation), $\cup$ (union) and $\ast$ (closure) operators
- $L_1$ denoted by RE $0^* \cup (0^* \circ 1 \circ 0^* \circ 1 \circ 0^*)^*$
- given RE $R$, the set of strings it denotes is $L(R)$
- pattern matching in text
- query languages for XML or RDF
Using Regular Expressions to Query Graphs

Graphs (networks) are widely used for representing data

- social networks
- transportation and other networks
- geographical information
- semistructured data
- (hyper)document structure
- semantic associations in criminal investigations
- bibliographic citation analysis
- pathways in biological processes
- knowledge representation (e.g. semantic web)
- program analysis
- workflow systems
- data provenance
- . . .
Using Regular Expressions to Query Graphs

- (my PhD thesis!)
- usually regular expressions used for string search
- consider data represented by a directed graph of labelled nodes and labelled edges
- regular expressions can express *paths* we are interested in
- sequence of edge labels rather than sequence of symbols (characters)
- a query using regular expression $R$ can ask for all nodes connected by a path whose concatenation of edge labels is in $L(R)$
Graph $G$ (where nodes represent people and places):

- $a$ is a citizenOf $SA$.
- $b$ is bornIn $CT$ and locatedIn $SA$.
- $c$ is bornIn $UK$ and livesIn $CT$.
Regular expression

\[ R = \text{citizenOf} | ((\text{bornIn} \mid \text{livesIn}) \cdot \text{locatedIn}^*) \]

asks for paths of edges between a person \( x \) and a place \( y \) such that:

- \( x \) is a citizenOf \( y \), or
- \( x \) is bornIn or livesIn \( y \), or
- \( x \) is bornIn or livesIn a place that is locatedIn \( y \)
Regular path query evaluation

**Regular Path Problem**

Given graph $G$, pair of nodes $x$ and $y$ and regular expression $R$, is there a path from $x$ to $y$ satisfying $R$?

**Algorithm:**

- construct a nondeterministic finite automaton (NFA) $M$ accepting $L(R)$
- assume $M$ has initial state $s_0$ and final state $s_f$
- consider $G$ as an NFA with initial state $x$ and final state $y$
- form the “intersection” (or “product”) $I$ of $M$ and $G$
- check if there is a path from $(s_0, x)$ to $(s_f, y)$

- Each step can be done in PTIME, so **Regular Path Problem** has PTIME complexity
NFA $M$ for $R = \text{citizenOf} \mid ((\text{bornIn} \mid \text{livesIn}) \cdot \text{locatedIn}^*)$
Intersection of $G$ and $M$
Intersection of $G$ and $M$
Intersection of $G$ and $M$
Intersection of $G$ and $M$
Intersection of $G$ and $M$
Intersection of $G$ and $M$
Regular simple path queries

- path $p$ is *simple* if no node is repeated on $p$
- **Regular Simple Path Problem**
  Given graph $G$, pair of nodes $x$ and $y$ and regular expression $R$, is there a *simple* path from $x$ to $y$ satisfying $R$?
Regular simple path queries

- path $p$ is \textit{simple} if no node is repeated on $p$

\textbf{Regular Simple Path Problem}

Given graph $G$, pair of nodes $x$ and $y$ and regular expression $R$, is there a \textit{simple} path from $x$ to $y$ satisfying $R$?

\textbf{Regular Simple Path Problem} is NP-complete [Mendelzon & Wood (1989)]
Regular simple path queries

- path $p$ is *simple* if no node is repeated on $p$

**Regular Simple Path Problem**

Given graph $G$, pair of nodes $x$ and $y$ and regular expression $R$, is there a *simple* path from $x$ to $y$ satisfying $R$?

**Regular Simple Path Problem** is NP-complete [Mendelzon & Wood (1989)]

- there can be a path from $x$ to $y$ satisfying $R$ but no simple path satisfying $R$, e.g., $R = (c \cdot d)^*$

![Diagram](attachment:image.png)
Approaches to deal with this problem

- what causes the problem?
Approaches to deal with this problem

- what causes the problem?
- the presence of cycles
Approaches to deal with this problem

- what causes the problem?
- the presence of cycles
- obvious first step is to consider graphs without cycles—DAGs
Approaches to deal with this problem

- what causes the problem?
- the presence of cycles
- obvious first step is to consider graphs without cycles—DAGs
- then might look at restricted forms of REs—we looked at those corresponding to languages closed under *abbreviations*
Approaches to deal with this problem

- what causes the problem?
- the presence of cycles
- obvious first step is to consider graphs without cycles—DAGs
- then might look at restricted forms of REs—we looked at those corresponding to languages closed under abbreviations
- then one might consider a combination of graphs and REs—we looked at graphs whose cycle structure does not conflict with the RE
Approaches to deal with this problem

▶ what causes the problem?
▶ the presence of cycles
▶ obvious first step is to consider graphs without cycles—DAGs
▶ then might look at restricted forms of REs—we looked at those corresponding to languages closed under abbreviations
▶ then one might consider a combination of graphs and REs—we looked at graphs whose cycle structure does not conflict with the RE
▶ finally showed that conflict-freedom is a generalisation:
  ▶ no RE conflicts with any DAG
  ▶ an RE closed under abbreviations never conflicts with any graph
Other approaches

- in general, may also run experiments to measure actual running times
Other approaches

- in general, may also run experiments to measure actual running times
- may also develop \textit{approximation} algorithms
  - can sometimes find a PTIME algorithm with a performance guarantee (e.g. for TSP, finds a tour at most twice the optimal distance)
  - other times this problem itself is NP-hard
Conclusion

- is my system/language more *powerful* than others?
- is my system/language more *efficient* than others?
- expressive power or computational complexity can be studied by relating them to
  - formal language theory: languages, grammars, automata, …
- tradeoff between expressive power and computational complexity
- consider restrictions of difficult problems or giving up exact solutions
References